

1 Data Strukturer

1.1 Linked List

Methods A `LinkedList` has the following methods

Method	Runtime
<code>add(E elem)</code>	$O(1)$ for all indices
<code>contains(E e)</code>	$O(n)$
<code>remove(E e)</code>	$O(n)$
<code>removeFirst(E e)</code>	$O(1)$
<code>removeLast(E e)</code>	$O(1)$
<code>toArray()</code>	$O(n)$
<code>indexOf(E e)</code>	$O(n)$
<code>get()</code>	$O(n)$
<code>set(int i, E e)</code>	$O(n)$

1.2 Array List

An `ArrayList` has the following methods

Method	Runtime
<code>add(int index, E elem)</code>	$O(n)$
<code>add(E elem)</code>	$O(n)$
<code>contains(E e)</code>	$O(n)$
<code>remove(E e)</code>	$O(n)$
<code>toArray()</code>	$O(n)$
<code>indexOf(E e)</code>	$O(n)$
<code>get</code>	$O(1)$
<code>set(int i, E e)</code>	$O(1)$

1.3 Heap (Priority Queue)

A heap, or Priority Queue, implements the following methods

Method	Runtime
<code>add(E e)</code>	$O(\log n)$
<code>offer(E e)</code>	$O(\log n)$
<code>remove(E e)</code>	$O(\log n)$
<code>poll()</code>	$O(\log n)$
<code>peek()</code>	$O(1)$
<code>element()</code>	$O(1)$
<code>contains(E e)</code>	$O(n)$

1.4 Hash-tabell (HashMap)

1.5 Binært Søketre

1.6 Union Find

1.7 Graf

2 Algoritmer

2.1 Søke Algoritmer

2.1.1 Binary Search

A binary search algorithm is a searching algorithm that finds the index of the element sent in as argument to the function. A prerequisite for the binary search algorithm is that the list we are performing the search on is **sorted**.

Runtime The runtime of the binary search algorithm is $O(\log(n))$.

Below is an implementation of the binary search algorithm.

```
public class BinarySearch<E extends Comparable<? super E>> {
    List<E> sorted;

    public BinarySearch(ArrayList<E> sorted) {
        this.sorted = sorted;
    }

    public int search(E target) {
        int lo = 0;
        int hi = sorted.size() - 1;
        int mid = 0;

        while (lo <= hi) {
            mid = hi - (hi - lo) / 2;

            if (sorted.get(mid).compareTo(target) < 0) {
                lo = mid + 1;
            } else if (sorted.get(mid).compareTo(target) > 0) {
                hi = mid - 1;
            } else {
                return mid;
            }
        }
        throw new NoSuchElementException("Element not in list");
    }
}
```

2.1.2 Quick Select

QuickSelect is a selection algorithm to find the *k*th smallest (or *k*th largest) element in a list. What's special about the **QuickSelect** algorithm, is that it is not necessary that the list we are selecting from is sorted. It is related to the **QuickSort** algorithm in that it selects a **pivot**, and partitions the list

based on this pivot. In contrast to the `QuickSort` algorithm, `QuickSelect` only recurses into one of the partitions of the list. It chooses the list which contains the `kth` element.

Runtime The `QuickSelect` algorithm has an average runtime of $O(n)$, and a worst case of $O(n^2)$. The worst case occurs when pivots are chosen poorly, such as only decreasing the pivot by one every time. This is easily mitigated by choosing a pivot in the of the current search scope at random.

Below is an implementation of the algorithm

```
public class QuickSelect<T extends Comparable<? super T>> {
    public T select(List<T> list, int left, int right, int k) {
        if (left == right) {
            return list.get(left);
        }

        Random rand = new Random();
        int pivot = rand.nextInt(right - left + 1);
        pivot = partition(list, left, right, pivot);

        if (k == pivot) {
            return list.get(k);
        } else if (k < pivot) {
            return select(list, left, pivot - 1, k);
        } else {
            return select(list, pivot + 1, right, k);
        }
    }

    private int partition(List<T> list, int left, int right, int pivot) {
        T pivotVal = list.get(pivot);
        Collections.swap(list, pivot, right);
        int storeIndex = left;

        for (int i = left; i < right; i++) {
            if (list.get(i).compareTo(pivotVal) < 0) {
                Collections.swap(list, storeIndex++, i);
            }
        }
        Collections.swap(list, right, storeIndex);
        return storeIndex;
    }
}
```

2.2 Sortering

2.2.1 Quick Sort

QuickSort is a sorting algorithm that operates on the Divide and Conquer principle. It is a sorting algorithm that can be done both in place, and not in place. It works by selecting a pivot in the list, and moves all elements that are smaller than the pivot to the left, and all that are greater to the right. Next, these lists are sorted recursively.

Runtime The QuickSort algorithm has a best-case runtime of $O(n \log n)$, and a worst case runtime of $O(n^2)$.

Below is an implementations of the QuickSort algorithm.

```
public class QuickSort <T extends Comparable<? super T>> {

    public List<T> sort(List<T> list) {
        int left = 0;
        int right = list.size() - 1;
        return qsort(list, left, right);
    }

    private List<T> qsort(List<T> list, int left, int right) {
        if (left >= right || left < 0) {
            return list;
        }

        int pivot = partition(list, left, right);

        qsort(list, left, pivot - 1);
        qsort(list, pivot + 1, right);

        return list;
    }

    private int partition(List<T> list, int left, int right) {
        T pivot = list.get(right);

        int tmp = left - 1;
        for (int i = left; i < right; i++) {
            if (list.get(i).compareTo(pivot) <= 0) {
                Collections.swap(list, ++tmp, i);
            }
        }

        Collections.swap(list, ++tmp, right);
        return tmp;
    }
}
```

2.2.2 Merge Sort**2.2.3 Insertion Sort****2.2.4 Selection Sort****2.3 Graf Algoritmer****2.3.1 Depth First Search****2.3.2 Breadth First Search****2.3.3 Minimum Spanning Tree****2.3.4 Dijkstra's Algorithm****2.3.5 Bellman-Ford**